Introduction

This Hydra game is designed solely by Yuchen Li (me) based on the given project description, “Hydra.pdf” (called Description in the remaining of this document). All code work is done independently by myself. All modifications on the original descriptions and enhancements (which will be listed in the following sessions) are initiated and accomplished by myself. The overall structure is mostly the same as the one on due date 1, but there are many extra functions/methods added. Most of them are added as private, and only a few of them are added as public.

Overview

The overall class structure can be found in the UML diagram. **Card** is the class for all cards, with a child class **Joker** which is for all Jokers. **Pile** is the class for all piles, including draw piles, discard piles and some other places (sometimes a method needs a temporary **pile** as a parameter. E.g. Reshuffle discard pile to form a new draw pile). Any player is stored in class **Player**. A *Player* has their own name (default to “Player x” where x is a unique integer beginning from 1, as specified in Description), draw pile, discard pile, current card, reserve card, etc. As an enhancement, **ComputerPlayer** is introduced as a child class of *Player*. A **ComputerPlayer** makes each decision on their own automatically and does not require any user input unless in testing mode. Any head is stored in class **Head**, which has its own index. To make operations easier to use (also for the purpose of designing toward OOP), **Heads** is a class containing a set of **Head**(s). New head(s) can be added to it, and old head(s) can be removed from it, as the Hydra game requires. Finally, the actual game is an instance of class **HydraBasic**, which is inherited from abstract class **Hydra**. The public interface of **HydraBasic** is very simple: a constructor with the only parameter being a bool variable indicating whether testing mode is enabled, and a method play() with no parameters which begins the game. Other methods are all private, which helps output messages, perform certain actions on heads, prompts actions from player(s), etc.

I would also like to mention the changes I decided to make on the game. The game flow and rules follow the specification in Description, while I chose to add some extra messages and some delay (1 second typically) between certain messages. I will call these “modifications” on the original description because they do not change any rules or any way of inputting a player’s move/choice. Modifications are added in order to give all players clearer views of what the current player is doing and what changes are being made on heads instead of popping up a bunch of messages at a time, which makes the next player (or even the current player) have no idea of what is going on. Upon receiving an invalid input (e.g. an invalid number of head as a move), instead of re-outputting the prompt message again, the program will output a message indicating your input is invalid (sometimes also the reason why it is invalid), and then re-prompt after a short delay. In addition to “modifications”, “enhancements” are also added to my project. There are 5 of them, which will be mentioned in Extra Credit Features. Before starting a game, the user will have an opportunity to choose any enhancement(s) they want to add to this game. Although the enhancement choices cannot be changed in middle of a game, the user can always start a new game with a new set of enhancement(s) they want.

Design

There are a number of design challenges I have met in this project. One of the major ones is the winning situation. A player can win at any time, and the game should stop immediately and announce the winner. This leads to my initial idea of using an Observer design pattern, where the player does "self-checking" on whether they have won, and if they do, notify their observer (which should be the game instance only). However, after actually trying it a little bit, it turned out that because all my "playing" process is enclosed in the method play(), even if the game is notified that someone has won, it would require some extra work to stop play(). For example, in method notify(...), I need to set some variable indicating someone has won, and in play(), I need to check whether that variable is set. With these extra checking to do with Observer design pattern, I decided it would be easier if I use an exception. When a player checks that they have won, throw the exception (in my program, it is PlayerWonException). The play() method is a big "try…catch" block. When PlayerWonException is caught, output the player's name (which is carried by the exception) and stop the game. In fact, there is another exception QuitGameException that is caught there, which simply quits the game. With this "exception catching" methodology, I can get rid of a lot of extra function invocations which check whether the current player has won, and therefore it makes my code cleaner and shorter. I also used the thought of "exception catching" at some other places, most of which are checking for invalid input.

Another design challenge is the enhancements. Thinking of how to implement enhancements, the first thing that came into my mind was Decorator design pattern. However, I do not think this is a typical problem that can be solved by Decorators. I believe the decorators are made to override functions so that each decorator has their own implementation, which is achievable in my design, but it's not worth it. The reason is that there may be a lot of functions that need to be overridden for some enhancement, and there would be so much code being copied over, which I think is not a good practice of OOP. Instead, I chose to use a vector of enhancements. At the beginning of each game, the user is asked to choose what enhancement(s) they want. Since the enhancements should not be changed during the game (it's not good to change the rules during a game), using a vector should be totally fine. To realize the enhancements, I added some extra code checking whether some specific enhancement is enabled, and if so, do something differently (e.g. if the grammar enhancement is enabled, check if the card value is 8 or A, and output "an" instead of "a" if so).

Last but not the least, the realization of **ComputerPlayer**. I decided to use pImpl in **Player** as a protected field and use methods *transferInfo(std::shared_ptr<PlayerImpl>)* and *acceptInfo(std::shared_ptr<PlayerImpl>)* to transfer information between an actual player and a computer player. The reason of using this rather than adding an accessor of pImpl is that I believe the accessor breaks the encapsulation a little bit. Using the two methods above, which is similar to a Visitor design pattern, ensures that the player's pImpl field is "retrieved" only when they are about to be replaced by another player. Also, using pImpl simplifies the parameter list to just one instead of all fields.

All of my class design aims to follow the concept of encapsulation, minimize coupling and maximize cohesion. All fields are private/protected, and accessors and/or setters are added only if they are necessary and do not break encapsulation.

Resilience to Change

I believe that if some rules are changed, the change to my code is minimal. I think for this Hydra game, the rule of what cards can/cannot be added to a head and what kind of move causes changes to the player's current round, in addition to the rule of cutting a head, are the main part of the game. Currently, each of these process/calculation is done at one place, which means if any of them changes, I will probably only need to find the one place (function) that performs the calculation and change it. For example, if the rule of adding card to a head is changed, say for option 1, instead of being able to put a card with value strictly less than that of the top card of the head, we need the value to be strictly larger, all we need to do is to change the sign from "<=" to ">=" at line 35 and the sign from "<" to ">" at line 37 of file "head.cc". A more complicated example would be that if we want to add a rule which says is we add a card with value J/Q/K on top of another with value J/Q/K, then it reduces the remaining number of card to be drawn by 1. Then we will add another checking before line 36 of "head.cc", which checks whether both the card that is about to be added and the top of the head are both one of J/Q/K, and set the return value to 5 if so (Since this function is invoked by "Heads::addCardToHead" located at line 38 of "heads.cc" and that function returns 4 to indicate that the player has to cut a head, we probably don't want to return 4 for Head::addCard(...), but this can totally be modified if we want). After that, at the end of line 406 of file "hydra_basic.cc", we need to add another scenario "else if (actionresult == 5) {remaining--;}"

Answers to Questions

Q1: What sort of design pattern should you use to structure your game classes so that changing the interface or changing the game rules would have as little impact on the code as possible? Explain how your classes fit this framework.

In the document submitted at due date 1, I said I would use Strategy pattern with an abstract base class **Hydra**. Although I still have that abstract base class with **HydraBasic** inherited from it, I no longer think that is necessary. Because of the way I design the game, the class **HydraBasic** invokes methods of **Player** and **Heads** (which invokes another method of **Head**) and let them do the calculations, **HydraBasic** only takes their results and do certain changes based on the results (e.g. finish a turn early). Therefore, when some rules are changed or added, we probably just need to change the calculation part, which is mostly done in some method of **Player** or **Head**, and change the action **HydraBasic** does based on the result.

Q2: Jokers have a different behaviour from any other card. How should you structure your card type to support this without special-casing jokers everywhere they are used?

[This answer is not changed from due date 1, as follows.] I use a class **Card** to represent all cards except jokers and a class **Joker** which inherit from class **Card** to represent jokers. Class **Joker** has its own public method which sets its card value to the specific value. This method can only be called once since as long as the value is chosen for a joker, it cannot be changed afterward. This way, I can use a (smart) pointer to **Card** to represent all cards and simply call **Joker**'s public method when needed without special-casing jokers due to C++'s dynamic dispatch.

Q3: If you want to allow computer players, in addition to human players, how might you structure your classes? Consider that different types of computer players might also have differing play strategies, and that strategies might change as the game progresses i.e. dynamically during the play of the game. How would that affect your structure?

[This answer is changed slightly from due date 1.] To allow computer players, I use a class **ComputerPlayer** which is inherited from the concrete class **Player**, which represents actual players. I decided to use just 1 type of **ComputerPlayer** instead of several types with different difficulties mostly because there is not much time for me to design strategies for computer players, but if I were to implement different difficulties, the Strategy design pattern would be the best choice. To make the changes on play strategies of computer players, I will implement a public method which changes the difficulty, and this will affect further decisions the computer player makes.

Q4: If a human player wanted to stop playing, but the other players wished to continue, it would be reasonable to replace them with a computer player. How might you structure your classes to allow an easy transfer of the information associated with the human player to the computer player?

[The main idea of the answer is not changed. Here is the modified answer.] First, I will use PImpl design pattern to make all private fields contained in a pointer to PImpl. To allow transfer of information between an actual **Player** and **ComputerPlayer**, I use methods *transferInfo(std::shared_ptr<PlayerImpl>)* and *acceptInfo(std::shared_ptr<PlayerImpl>)* to transfer information between them. The advantage of using PImpl is that we don't have to change the signature of the "accepting" method of **ComputerPlayer** when some fields of **Player** change in the future.

Extra Credit Features

1. **Enhancement – Grammar**. As suggested in Description, the game says you're holding "a A" or "a 8" instead of "an". With this enhancement enabled, this grammar error will be fixed.
2. **Enhancement – House Rule**. As part of the suggestion in Description, with this enhancement enabled, a player can cut a head even if cutting a head is not the only option.
3. **Enhancement – Customized Names**. With this enhancement enabled, each player can enter their own unique name before the game starts. The players will be displayed with their chosen name rather than the default "Player 1, Player 2, etc.", although they can choose to have a name like that by entering them.
4. **Enhancement – Computer Players**. With this enhancement enabled, the game allows some player(s) to be disconnected from the game and let a naïve computer player play for them, and it allows some disconnected player(s) to be reconnected to the game to continue playing (replacing the computer player in their original place). This action is only available between 2 adjacent turns

(i.e. when notifying "xxx, it is your turn.") since you probably don't want to interrupt someone else's turn.

5. **Enhancement – Smaller decks**. With this enhancement enabled, the game will use 8-card decks instead of normal 54-card decks. An 8-card deck contains AS, 4S, 8S, AD, 4D, 8D, Joker, Joker.
6. **Modification – More detailed messages, invalid input notification and delay between actions**. I am not sure whether this counts towards Extra Credit Features since it is not an enhancement (as discussed in section Overview), but I believe it is worth mentioning. The messages are output with more detailed information (e.g. "Card 8S is added successfully to head 4!"), invalid inputs will cause an error message being output indicating that was an invalid input and sometimes with reasons (e.g. "You entered an invalid index of head. A valid index is between 3 and 11. Please enter again."), and there are delays between actions to give all players clearer view of what is going on with game (e.g. "Player 1 is cutting a head…" - 1s delay - "New heads will be formed by 9C \n 2H" - 1s delay – "Player 1 has successfully cut a head!" - 1s delay - …)
7. **All pointers are smart pointers**. No raw pointers are used. No "new" or "delete" keywords are used. All pointers are smart pointers.

Final Questions

1. What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

   Writing large programs requires a lot of work on many different aspects. The first is designing the structure and deciding which design patterns to use if any. This can take a lot of time since some of them may not be clear until I really dig into it or even so some actual code work. I believe as I design more projects and participate in more project work, designing will become easier and clearer for me. Another lesson is that starting from small is really important and useful. For example, as I design **Card** and **Pile**, I realize that there are some extra functions that are needed for the "bigger" class **Head**. Then I will take notes that I will add those functions later when I implement **Head**. This can save me a lot of time of getting back to the previous class and add another function.

2. What would you have done differently if you had the chance to start over?

   I would definitely spend more time before due date 1 on designing the structure and the functions I need. It turns out that I changed a lot of public interfaces from those in the UML I submitted on due date 1. If that original UML was intact and perfect, it would have saved me a lot of time implementing the classes.

Conclusion

The basic Hydra game is intact and playable. The game meets all requirements that are needed for a basic Hydra game as specified in Description. However, there are lots of add-ons and enhancements that I can do on top of it. I will keep improving the game and adding more features in the future.